Analyse d'une Session BitTorrent

- (fetching news ...)

sources: [1]The BitTorrent Protocol Specification, [2]Tracker Returns Compact Peer Lists, [3]http://peertech.org/BittorrentProtocolSpec, [4]http://wiki.theory.org/index.php/BitTorrentSpecification

---

1. Récupérer le fichier .torrent

Il s'agit ici d'un banal téléchargement HTTP, si ce n'est que le fichier récupéré aura le type mime "application/x-bittorrent". Ce fichier contient les informations nécessaires pour identifier le fichier et démarrer son téléchargement sous la forme d'un *dictionnaire b-encodé* (bencoded messages, un concept hérité du langage "Python").

```
wget http://torrent.linux.duke.edu/specifix.torrent
```


```
d8:announce43:http://torrent.linux.duke.edu:6969/announce1
3:creation datei1089948866e4:infod5:filesld6:lengthi159332 e4:path16:conary18:conary-
0.1.tar.bz2eed6:lengthi57e4:path
15:linux5:alpha3:0.13:iso6:MD5SUMeed6:lengthi1778e4:path15
:linux5:alpha3:0.13:iso6:READMEeed6:lengthi618993664e4:pat
h15:linux5:alpha3:0.13:iso22:specifix-linux-0.1.isoeee4:na
me8:specifix12:piece lengthi262144e6:pieces47240:uÚÎ°2D:×í
Î¡1bõJu(...)
```


le contenu du fichier .torrent (extrait), les clés sont en gras, les valeurs soulignées.

Le b-encodage permet la représenation de structures complexes (listes et dictionnaires imbriqués) à partir de règles simples:

- une chaîne est préfixée par sa longueur et le symbole ':' (ex `12:Hello world!` est la chaîne "Hello world!")
- un nombre est précédé du symbole 'i' et terminé par le symbole 'e'. Tous les nombres sont représentés en base 10 (ex: `i65536e` est le nombre 2^16)
- une liste est précédée du symbole 'l' et terminée par le symbole 'e'. Aucun symbole spécifique ne sépare les éléments de la liste. (ex: `l5:linux5:alpha3:0.13:iso6:READMEe` est la liste des chaînes [linux, alpha, 0.1, iso, README]). N'importe quel type peut être utilisé pour les éléments (chaîne, nombre, liste, dictionnaire, etc.). 'le' représente une liste vide.
- un dictionnaire est précédé du symbole 'd' et suivi par le symbole 'e'. Chaque entrée du dictionnaire est un couple formé par une chaîne (la clé) puis sa valeur(tous les types sont autorisés).

En appliquant le décodage be (cf bibliothèque fournie en annexe), on trouve la structure suivante

- **announce**=>http://torrent.linux.duke.edu:6969/announce
- **creation date=>**1089948866
- **info=>***<dictionnaire>* **{**
  - **files=>***<liste>***[**
    - *<dictionnaire>* **{ length=>**159332; **path=>** [conary, conary-0.1.tar.bz2] **}**
    - **{ length=>**57; **path**=>[linux, alpha, 0.1, iso, MD5SUM] **}**
    - **{ length=>**1778; **path**=>[linux, alpha, 0.1, iso, README] **}**
    - **{ length=>**618993664; **path=>[**linux, alpha,0.1, iso, specifix-linux-0.1.iso**]}**
    - **]**
  - **name=>**specifix
  - **piece length=>**262144
  - **pieces=>**47240:uÚÎº2D:×íÎ¡1bõJu(...)
  - **}**

Le fichier `specifix.torrent` décrit donc une archive de 4 fichiers découpées en 2362 tronçons (*piece*) de 262144 octets chacuns. Le *tracker* à contacter pour se connecter aux fournisseurs de cette archive est joignable via l'URL *http://torrent.linux.duke.edu:6969/announce*. Notez qu'il est possible que l'URL ne contienne pas de numéro de port. Dans ce cas, c'est le port par défaut de HTTP qui sera utilisé: le port 80 ...

2. Contacter le Tracker

Dans la terminologie BitTorrent, un *tracker* est une machine que chaque *peer* contacte pour connaître l'adresse d'autres *peers* occupés à télécharger le même fichier. Le tracker se comporte comme un serveur HTTP auquel il est nécessaire de passer un certain nombre de paramètres:

```
wget http://torrent.linux.duke.edu:6969/announce ?
info_hash=%c9%60%ffi%bc9%11%dc%96M%a69%7e%c7a%11%acR%b0%24
&peer_id=somethingelse
&ip=asmodan.run.montefiore.ulg.ac.be
&port=6881
```

La valeur de *info_hash* est le résultat du hashage de la valeur de *info* dans le fichier .torrent (ç.a.d. l'application de `sha1sum` sur le fichier dont on a extrait:
"`d8:announce43:http://torrent.linux.duke.edu:6969/announce13:creation dateai1089948866e4:info`" ainsi que le "**e**" final

```
tp> sha1sum specifix.torrent.nfo
c960ff69bc3911dc964da6397ec76111ac52b024  specifix.torrent.nfo
```

sha1sum renvoie la valeur sous la forme d'une chaîne hexadécimale. Pour que cette chaîne soit 'lisible' par le serveur HTTP, il est nécessaire d'*échapper* chacun des caractères (le *i*ème caractère du jeu est représenté alors par le symbole `%` suivi de la valeur hexadécimale de ce caractère). En d'autre terme, puisque `'A'==0x41`, le symbole `A` serait échappé en %41.
Certains caractères tels que A == %41, i==%69, 9=%39, etc (en fait tous les *alphanum*) pourraient être transmis sans échappement (caractères en gras dans l'exemple ci-dessus), mais échapper tous les caractères est probablement le plus simple dans ce cas.

work/tp> **telnet torrent.linux.duke.edu 6969** Trying 152.3.183.77... Connected to torrent.linux.duke.edu. Escape character is '^]'.	Contactons le tracker avec l'adresse et le port renseignés par 'announce'...
GET http://torrent.linux.duke.edu:6969/announce**?info_hash=%c9%60%ff%69%bc%39%11%dc%96%4d%a6%39%7e%c7%61%11%ac%52%b0%24&**peer_id=xxxxyyyyxxxxyyyxxxx**&ip=**139.165.223.16**&port=**6881**&uploaded=**0**&downloaded=**0**&left=**61899**3664 HTTP/1.1<CRLF> Host: torrent.linux.duke.edu:6969<CRLF> <CRLF>	La requête transmise (normalement, elle ne prend que deux lignes)
HTTP/1.0 200 OK<CRLF> Content-Length: 286<CRLF> Content-Type: text/plain<CRLF> Pragma: no-cache<CRLF> <CRLF>	L'en-tête de la réponse du tracker, toujours respectant le protocole HTTP
d --8:interval=>i**1800**e --5:peers=>l ----d ------2:ip=>11:**83.72.54.24** ------7:peer id=>20:**M3-4-2--2cd992318c8e** ------4:port=>i**6882**e ----e ----d ------2:ip=>13:**80.15.205.190** ------7:peer id=>20:**-AZ2104-lN5SvwK0XgRt** ------4:port=>i**92**e ----e	Le contenu de la réponse est un fichier b-encodé (ici légèrement remanié pour être un peu plus lisible).
----d ------2:ip=>11:**15.23.13.77** ------7:peer id=>20:**4xJŸä^ïD** ------4:port=>i**6881**e ----e	Chaque entrée de la liste *peers* décrit une des machines possédant des morceaux du fichier.
----d ------2:ip=>11:**24.26.56.46** ------7:peer id=>20:**-AZ2104-eSDSTnXkJ2In** ------4:port=>i**6881**e ----e --e e	
Connection closed by foreign host.	

Note: Le *tracker* va nous annoncer un intervalle de temps avant lequel il ne désire pas être réinterrogé. Il y a de fortes chances pour que cet intervalle soit plus long que le *timeout* sur la connexion HTTP. Il est donc préférable, dans ce cas-ci, de ne pas faire de *pipelining* mais d'envoyer l'option `Connection: close` dans l'en-tête de la requête. On sera alors certain que le *tracker* fermera la connexion *juste après avoir envoyé les derniers bytes* de sa réponse.

Nous avons donc à l'heure actuelle 4 machines proposant des morceaux du fichier qui nous intéresse: 83.72.54.24 (port 6882), 80.15.205.190(port 92), 15.23.13.77 (port 6881) et 24.26.56.46 (port 6881). On peut se connecter à n'importe laquelle d'entre-elles par le biais d'un *telnet*.

Compact Peer List

Depuis 2008, un tracker peut opter, pour économiser la bande passante pour une *version compacte de la liste des peers.* Il est possible de demander expressément cette version compacte en ajoutant *&compact=1* dans la requête ou de demander la version étendue en ajoutant *&compact=0*. Cette extension du protocole initiale est décrite dans le BEP#23 [2]. A noter que certains trackers ne supporteront que l'une des deux et on pourrait recevoir un message d'erreur si l'on contacte un tracker ne supportant que la version compacte sans paramètre "compact=x".

```
GET /announce?info_hash=...&peer_id=...&event=started HTPP/1.1
Host: torrent.ubuntu.com

HTTP/1.0 400 Bad Request
Content-Length: 39
Content-Type: text/plain

your client is outdated, please upgrade
```

On peut facilement détecter que le tracker a employé une liste compacte par le fait que la clé *peers* de sa réponse contient une *chaîne b-encodée* et non pas une *liste*. Cette chaîne doit être traitée comme un tableau de bytes "raw" (de la même manière que le SHA) et est constituée de n éléments de 6 bytes chacun (4 bytes d'adresse IP + 2 bytes de port, au Network Byte Ordering)

Remarque: les adresses présentes ici sont purement fictives

La bibliothèque *openSSL* offre de nombreuses fonctions de hashage, dont *SHA1*. Consultez la page de manuel (man sha -S3) pour plus d'info et utilisez `gcc -lssl` lors de la compilation.

3. Connexion à un peer

Une fois les valeurs de 'ip' et 'port' récupérées, et une fois la connexion établie avec le *peer*, on quitte le protocole HTTP. Chacune des partie va envoyer sur la connexion un *handshake* de 68 bytes (dans la version actuelle du protocole) comportant le hashid et son peerid

contenu	description	long.
`\23BitTorrent protocol`	chaîne identifiant le protocole, précédée par sa taille (i.e. 19 bytes)	20
`\0\0\0\0\0\0\0\0`	Les extensions de protocole utilisées (ici, aucune extension supportée) (cf. http://bittorrent.org/beps/bep_0004.html)	8
sha1sum des infos	la signature du fichier pour cette session (envoyée au Tracker) en format binaire. A envoyer comme un tableau de bytes. (Attention, une conversion vers/du Network Byte Order est nécessaire si vous travaillez *mot par mot* à la place ...)	20
peerid	identificateur de l'entité qui transmet (même format que le sha1sum)	20

Si la signature du protocole est incorrecte ou que le *sha1sum* ne correspond pas avec ce qui est attendu, la connexion sera automatiquement fermée.

Une fois le *handshake* terminé, un peer qui possède déjà des morceaux de fichier est tenu d'envoyer un message *bitfield* en premier lieu (cf section suivante)

4. Le protocole entre peers

Tous les messages qui suivent utilisent le *network byte order*. Sauf contre-indication, tous les nombres sont codés sur 32 bits. Consultez [2,4] pour des informations plus détaillées sur le choking. Tous les messages commencent par un entier donnant leur taille puis un byte indiquant le type de message.

4.1 accueil d'un nouveau téléchargeur

On est ici dans le cas ou un *peer P* qui n'a encore aucun morceau du fichier contacte un *peer Q* qui possède du contenu intéressant pour *P* et qui est disposé à l'offrir:

1. *Q* transmet un message *bitfield* (type 5) indiquant quels tronçons il possède, **<len=X+1><05><bi><tf><ie><ld><ta><il><le><=X>**. Chaque bit du bitfield représente un tronçon (1° byte & 0x80 == disponibilité du 1° tronçon)
2. *P* transmet un message *interrested* (type 2) pour signaler qu'il désire recevoir des données depuis *Q*. **<00000001><02>**
3. Dès que *Q* est près à transmettre, il envoie le message *unchoke* **<00000001><01>**

4. *P* peut maintenant demander des données. chaque message *request* doit indiquer le n° du tronçon concerné (*index* dans le *bitfield*), et un intervalle de bytes de ce tronçon à transmettre (*offset* et *longueur*). Le protocole recommande des requêtes de 32K bytes. Les messages à envoyer successivement pour récupérer l'entièreté du 3° tronçon de 256K seraient donc:
   - --> *request* **<0000000d><06><00000003><00000000><00008000>**
   - <-- *piece* **<00008009><07><00000003><00000000>** ***32K de données***
   - --> *request* **<0000000d><06><00000003><00008000><00008000>**
   - <-- *piece* **<00008009><07><00000003><00008000>** ***32K de données***
   - --> *request* **<0000000d><06><00000003><00010000><00008000>**
   - **...**
   - <-- *piece* **<00008009><07><00000003><00038000>** ***32K de données***

4.2 un tronçon a été complètement reçu

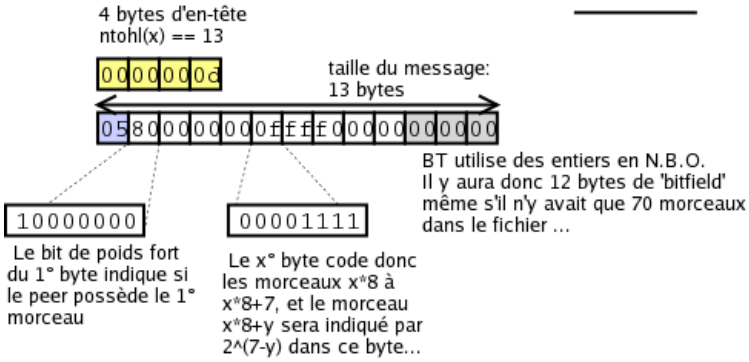
Nous sommes ici dans le cas où un *peer P* vient de recevoir le dernier bloc complétant le tronçon #i alors qu'il est connecté aux *peers Q1..Qn*.

1. *P* va calculer le hash SHA1 du tronçon reçu et s'assurer qu'il correspond bien au hash stocké dans le fichier info, c.à.d. aux bytes `[i*20..19+i*20]` de la chaîne `info::pieces`
2. Si les données concordent avec le hashage annoncé, *P* peut ajouter les données au fichier récupéré de manière permanente.
3. *P* annonce à ses voisins qu'il possède maintenant le tronçon numéro *i* à l'aide du message *have*. **<00000005><04><iiiiiiii>**
4. Les voisins de *P* vont, suite à cette annonce, signaler leur (dés)intérêt par un des messages *interested* **<00000001><02>** ou *not interested* **<00000001><03>**.
5. S i *P* est disposé à accepter de nouvelles connexions en *upload*, il transmettra un message *unchoke* **<00000001><01>** en réponse à *interested*. Le message *choke* **<00000001><00>** lui permet au contraire de mettre en attente un *peer*.

5. Quelques précisions sur les "bitfields"

5.1 Le B.A.BA des bitmaps

Le principe d'un bitfield (ou bitmap, ce qui est plutôt le cas, ici) est simple: une chaîne de bits ou chaque bit indique la présence ou l'absence d'un élément. Dans BitTorrent, le bitfield est transmis comme une succession de *N* entiers ***non-signés*** de 32 bits au *Network Byte Ordering.* La figure ci-dessous présente le message qu'aurait envoyé un *peer* possédant le morceau n° **0** (le premier) et les morceaux **36** à **41**.



On constatera que la taille du message contient toujours bien une *taille en bytes*. Le nombre de morceaux constituant l'archive, lui, peut être déduit du fichier `.torrent` (en divisant la taille de *pieces* par 20, puisque cette chaîne contient un SHA1 par morceau ;)

Vous pouvez "visualiser" le bitfield assez simplement avec un code du genre

```
unsigned char bitfield[];
for (i=0;i<msg_len;i++) printf("%02x",bitfield[i]);
```

5.2 Les opérations de base sur un bitmap

Evidemment, pour pouvoir manipuler indépendamment les bits, il faut un peu plus de chipotage. L'idée est d'utiliser les opérateurs "bit-à-bit" du langage C, c'est à dire l'opérateur "and": **`x&mask`** pour isoler un (ou des) bits et l'opérateur "or": **`x|mask`** pour mettre à 1 certains bits. Ainsi on peut tester individuellement les bits (dans l'ordre "de gauche à droite") avec

les constantes `0x80`, `0x40`, `0x20`, `0x10`, `0x08`, `0x04`, `0x02`, `0x01`.

En d'autres termes, vous pouvez tester individuellement chaque bit avec une boucle

```
for (mask=0x80;mask;mask=mask/2) {
   if (bitfield[i]&mask) printf("un");
   else printf("zero");
}
```

Un peu plus complexe, mais tout aussi intéressant, vous pouvez déterminer si le *k°* bit de tout le bitfield est présent avec

```
{
   unsigned i=k/8;
   unsigned char mask=1<<(7-(k%8));
   if (bitfield[i]&mask)
      printf("morceau %i présent",k);
}
```

En effet, si on réécrit k=i*8+j, on se rend compte que k/8 (division entière) donne le numéro du byte à tester et k%8 (reste de la division entière -- modulo) le numéro du bit à l'intérieur de ce byte, sachant que BitTorrent compte le bit le plus significatif d'abord. L'expression `1<<j` décale la constante 1 de *j* positions vers la gauche -- ou de multiplier 1 *j* fois par 2. *Cette opération est aussi performante qu'une addition sur un processeur.*

Si d'aventure nous avions voulu *modifier* l'état du k° morceau, on pouvait remplacer le 'if(...) printf(...)' par

`bitfield[i]=bitfield[i]|mask;` // mise à 1

ou

`bitfield[i]=bitfield[i]&(~mask);` // remise à 0

où `~x` inverse tous les bits d'un nombres (on extrait donc tous les bits sauf celui qui nous intéresse, qui reste à 0)

5.3 Compter et chercher

Si l'on réutilise froidement les opérations de bases dans des boucles, les résultats sont généralement catastrophique. Il existe quelques *"bons trucs"* bien connus des geeks adeptes de l'assembleur qui permettent de se faciliter la vie, notamment en exploitant le fait qu'en divisant successivement par 2, on finit par avoir vu tous les bits à 1 dans le bit de poids le plus faible.

Imaginons que nous devions compter le nombre de bits dans 25 ...

25	impair -> 1 bit
12	pair -> tjs 1 bit
6	pair -> tjs 1 bit
3	impair -> 2 bits
1	impair -> 3 bits

Il y a en effet 3 bits dans 25 (11001). En C, nous aurions écrit

```
for (x=25,nbits=0;x>0;x=x/2) {
  if (x&1) nbits++;
}
```

Au passage, remarquons que si aucun des bits n'est à 1, on s'en rend compte de suite, de même que l'on s'arrête dès que tous les bits à 1 ont été "consommés". S'il y a de fortes chances que tous les bits d'un byte soient simultanément à 1, on peut facilement accélérer le comptage des bits d'une longue chaîne en tenant compte du fait que `0xff` contient 8 bits (et en augmentant directement *nbits* de 8 et en passant au suivant). Il est même possible d'accélérer encore les opérations en manipulant les données *entier par entier* plutôt que caractère par caratctère (si `ix==0xffffffff`, on augmente nbits de 32 d'un coup).

En utilisant le même principe, on peut facilement rechercher le 1° bit à 0 à partir d'une position donnée:

// V et I sont définis par la position de départ. v=i=0 pour scanner tout le bitmap.

```
while (i<bf_length) {
  for (;v>0;v=v/2,j++)
    if ((v&1)==0) break;
  while (i<bf_length && bitfield[i]==0xff) i++;
  v=bitfield[i]; j=0;
}
```

*// si i<bf_length, i*8+j donne la position du bit trouvé.*

Attention: les opérateurs **&**, **|**, **<<** on des priorités un peu curieuses pour le non-averti. En particulier, *&* intervient *après* **==**, donc `(v&1==0)` est identique à `(v&(1==0))` et non à `((v&1)==0)`. Idem pour << et +: 4*2+1 vaut 9 alors que 4<<1+1 vaut 16. En cas de doutes, enrobez-les de parenthèses pour être sûr que les choses se déroulent dans l'ordre prévu :P

Erreurs Courantes

Le tracker refuse de me répondre

assurez-vous que le message envoyé est correct. En particulier, certains trackers ne sont pas HTTP/1.1 et ils ne supportent pas la présence du nom de domaine dans l'url. (`/announce?...` et pas `www.tracker.org/announce?...`)

Le tracker répond, mais avec en annonçant "no support for this torrent"

Cette erreur est généralement dûe à une erreur dans le SHA transmis. Un outil comme`btshowmetainfo.py` du package BitTorrent peut vous donner le hash exact. Gardez à l'esprit qu'il peut y avoir des champs "comment", etc. auxquels vous n'auriez pas pensé dans le fichier .torrent ...

Les peers que je contacte ne terminent pas le handshake

Il est fort possible qu'ils attendent une réponse de votre part avant d'envoyer la leur. Assurez-vous que vous avez au moins envoyé la chaine "BitTorrent Protocol" et votre propre ID avant d'essayer de lire l'ID du fichier venant de votre peer.

Les peers qui *me* contactent ne terminent pas le handshake

Celà arrive principalement si vous quittez un torrent puis en ouvrez un autre avec le même port. Les peers de l'ancien torrent mettront un certain temps à apprendre que vous n'êtes plus là et durant ce laps de temps, ils essayent de vous contacter, constatent que l'ID du fichier n'est pas celui attendu et se déconnectent. Vous pouvez envoyer une requête au tracker en ajoutant "`&event=stopped`" avant de quitter un torrent pour éviter ce genre de désagrément.

Tous les peers finissent par partir au bout de quelques minutes

C'est probablement dû à un non-respect du protocole de votre part. Si vous ne voyez rien d'anormal dans les messages échangés, assurez-vous d'avoir répondu par un "keep-alive" aux messages "keep-alive" reçus. Dans le cas contraire, le peer peut considérer que vous êtes plantés et finir par partir.

La ManPage de SHA1

sha(3) OpenSSL sha(3)

NAME

SHA1, SHA1_Init, SHA1_Update, SHA1_Final - Secure Hash Algorithm

SYNOPSIS

```
#include <openssl/sha.h>

   unsigned char *SHA1(const unsigned char *d, unsigned long n,
                       unsigned char *md);

   void SHA1_Init(SHA_CTX *c);
   void SHA1_Update(SHA_CTX *c, const void *data,
                       unsigned long len);
   void SHA1_Final(unsigned char *md, SHA_CTX *c);
```

DESCRIPTION

SHA-1 (Secure Hash Algorithm) is a cryptographic hash function with a 160 bit output.

SHA1() computes the SHA-1 message digest of the n bytes at d and places it in md (which must have space for SHA_DIGEST_LENGTH == 20 bytes of output). If md is NULL, the digest is placed in a static array.

The following functions may be used if the message is not completely stored in memory:

SHA1_Init() initializes a SHA_CTX structure.

SHA1_Update() can be called repeatedly with chunks of the message to be hashed (len bytes at data).

SHA1_Final() places the message digest in md, which must have space for SHA_DIGEST_LENGTH == 20 bytes of output, and erases the SHA_CTX.

Applications should use the higher level functions EVP_DigestInit(3) etc. instead of calling the hash functions directly.

The predecessor of SHA-1, SHA, is also implemented, but it should be used only when backward compatibility is required.

RETURN VALUES

SHA1() returns a pointer to the hash value.

SHA1_Init(), SHA1_Update() and SHA1_Final() do not return values.

CONFORMING TO

SHA: US Federal Information Processing Standard FIPS PUB 180 (Secure Hash Standard), SHA-1: US Federal Information Processing Standard FIPS PUB 180-1 (Secure Hash Standard), ANSI X9.30

SEE ALSO

ripemd(3), hmac(3), EVP_DigestInit(3)